Multisource Rumor Spreading with Network Coding

Yérom-David Bromberg
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
david.bromberg@irisa.fr

Quentin Dufour
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
quentin.dufour@inria.fr

Davide Frey
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
davide.frey@inria.fr

***Abstract*—The last decade has witnessed of a rising surge interest in Gossip protocols in distributed systems. In particular, as soon as there is a need to disseminate events, they become a key functional building block due to their scalability, robustness and fault tolerance under high churn. However, Gossip protocols are known to be bandwidth intensive. A huge amount of algorithms has been studied to limit the number of exchanged messages using different combination of push/pull approaches. We are revisiting the state of the art by applying Random Linear Network Coding to further increase performances. In particular, the originality of our approach is to combine sparse vector encoding to send our network coding coefficients and Lamport timestamps to split messages in generations in order to provide an efficient gossiping. Our results demonstrate that we are able to drastically reduce the bandwidth overhead and the delay compared to the state of the art.**

***Index Terms*—network coding, gossip-based dissemination, peer-to-peer network, epidemic algorithm, rumor spreading**

I. Introduction

Distributed systems are becoming increasing large and complex, with an ever-growing shift in their scale and dynamics. This trend leads to revisit key known challenges such as inconsistencies across distributed systems due to an inherent increase of unpredictable failures. To be highly resilient to such failures, an efficient and robust data dissemination protocol is the cornerstone of any distributed systems. The last decade gossip protocols, also named epidemics protocols, have been widely adopted as a key functional building block to build distributed systems. For instance, gossip protocols have been used for overlay construction [1], [2], membership and failure detection [3]–[6], aggregating data [7] and live streaming [8], [9]. This wide adoption comes from the resilience of gossip protocols while being simple and naturally distributed [10] [3] [11]. Gossip based dissemination can be simply represented as the random phone call problem; at the beginning, someone learns a rumor, and calls a set of random friends to propagate it. As soon as someone learns a new rumor, in turn, she randomly propagates it to her own set of friends, and so on recursively. Further, depending on whether there are one or more sources of rumors (i.e. dissemination of one or multiple messages to all nodes), gossip protocols may be either single or multi source. In both cases, randomness and recursive probabilistic exchanges provide scalability, robustness and fault tolerance under high churn to disseminate data while staying simple.

To appear in Proc. IEEE INFOCOM 2019.

However, due to its probabilistic aspect, gossip based dissemination implies high redundancy with nodes receiving the same message several times. Many algorithms have been studied to limit the number of exchanged messages to disseminate data, using different combination of approaches such as *push* (a node can push a message it knows to its neighbors), *pull* (a node pulls a messages it does not know from its neighbors) or *push-pull* (a mix of both) for either single or multi-source gossip protocols [12] [13] [14].

In this paper, we make a significant step beyond these protocols, and provide better performance with respect to the state of the art of multi-source gossip protocols.

The key principle of our approach is to consider the redundancy as a key advantage rather than as a shortcoming by leveraging on the Random Linear Network Coding (RNLC) techniques to provide efficient multi-source gossip based dissemination. Indeed, it has been shown that RLNC improves the theoretical stopping time, i.e., the number of rounds until protocol completeness, by sending linear combination of several messages instead of a given plain message, which increases the probability of propagating something new to recipients [15], [16].

Unfortunately, applying RNLC to multi-source gossip protocols is not without issues, and three key challenges remain open. First, existing approaches suppose that a vector, where each index identifies a message with its associated coefficient as a value, is disseminated. This approach implies a small namespace. In the context of multi source, the only option is to choose a random identifier over a sufficiently large namespace to have a negligible collision probability. However this does not scale. Some algorithms provide a mechanism to rename messages to a smaller namespace [17], but this kind of techniques are not applicable to gossip protocols as they would substantially increase the number of exchanged messages, and inherently the delay. Second, to reduce the complexity of the decoding process, messages are split in groups named generations. Existing rules to create generations require having only one sender, which is impractical in the context of multiple sources. Third, the use of RNLC implies linear combinations of multiple messages. This leads to potential partial knowledge of messages received, making precise message pull requests useless and breaks pull frequency adjustment based on missing messages count.

In this paper, we introduce CHEPIN, a CHeaper EPidemic dissemINation approach for multi-source gossip protocols.

To the best of our knowledge, our approach is the first one to apply RNLC to multi-source gossip protocols, while overcoming all the inherent challenges involved by the use of network-coding techniques.

More precisely, we make the following contributions.

- We solve the identifier namespace size *via* the use of sparse vectors. The data sent over the network stays reasonable.
- We create generations for messages from multiple sources by leveraging on Lamport timestamps. All messages sharing the same clock are in the same generation whatever its source.
- We overcome the issue of partial message knowledge by providing an adaptation of push, pull, push-pull gossip protocols. We pull specific generations instead of specific messages.
- We introduce updated algorithms to make our approach applicable to the current state of the art of multi-source gossip protocols.
- Finally, we are evaluating CHEPIN thoroughly by simulation. We show that our solution reduces the bandwidth overhead by 25% and the delivery delay by 18% with respect to PULP [14], while keeping the same properties.

II. Related work

Epidemic protocols were introduced in 1988 by Xerox researchers on replicated databases [18]. They introduce three protocols to exchange rumors: push, pull and push pull.

a) Push protocols: to transmit rumors, push-based protocols imply that informed nodes relay the message to their neighbors. Some protocols are active, as they have a background thread that regularly retransmits received rumors, like balls and bins [12]. But it can also be implemented as a reactive protocols, where rumors are directly forwarded to the node's neighbors upon reception, like infect and die and infect forever protocols [13]. Push protocols are particularly efficient to quickly reach most of the network, however reaching all the nodes takes more time and involves significant redundancy, and thus bandwidth consumption.

b) Pull protocols: nodes that miss messages ask other nodes for the missing messages. As a consequence, *pull protocols* more efficiently reach the last nodes of the network, as inherently, they get messages with higher probability. However, they require sending more messages over the network: (i) one to ask a missing message, and (ii) one other for the reply that contains the missing message. Furthermore a mechanism or a rule is needed to know what are the missing messages to pull, which explains why these protocols are generally used in conjunction with a push phase.

c) Push-Pull protocols: the aim is to conciliate the best from push and pull protocols by reaching as many nodes as possible without redundancy on the push phase. Then, nodes that have not received the message will send pull requests to other nodes of the network. By ordering messages, Interleave [19] proposes a solution to discover the missing messages in the pull phase, but works only with a single source. Instead of ordering messages, Pulp [14] piggybacks a list of recently received messages identifiers in every sent messages, allowing multiple sources.

Gossip-based disseminations are caracterized by the reception of many redundant messages in the push and pull phases in order to receive every message with high probability. By using Random Linear Network Coding, we aim at increasing the usefulness of exchanged messages.

d) Random Linear Network Coding: To improve dissemination, some protocols use erasure coding [20] [19] or Random Linear Network Coding [21] but need encoding at the source or message ordering, which limits these techniques to single-source scenarios. Theoretical bounds have also been studied for multi-sender scenarios [15] [16] but they do not consider generations and suppose that messages are previously ordered. Network Coding is also used on wireless networks [22] [23] but with different system models, as emitted messages can be received by every node within range.

Applying RLNC gossip in a multi-sender scenario implies determining to which generation a message will belong to without additional coordination, and finding a way to link network coding coefficients to their respective original messages inside a generation.

III. Background on network coding and challenges

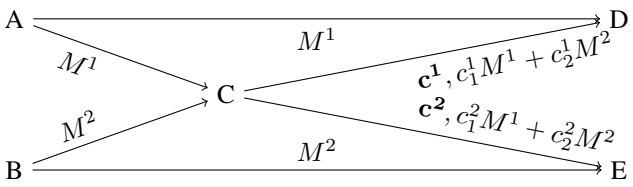


Fig. 1: With RLNC, C can send useful information to D and E without knowing what they have received

RLNC [24] provides a way to combine different messages on a network to improve their dissemination speed by increasing the chance that receiving nodes learn something new. In Figure 1, node C cannot know what D and E have received. By sending a linear combination of M^1 and M^2, nodes D and E can respectively recover M^2 and M^1 with the help of the plain message they also received. Without RLNC, node C would have to send both M^1 and M^2 to D and E involving two more messages. Every message must have the same size, defined as L bits thereafter. To handle messages of different size, it is possible to split or pad the message to have a final size of L bits.

The message content has to be split as symbols over a field $\mathbb{F}_{2^n}$. The $\mathbb{F}_{2^8}$ field has interesting properties when doing RLNC. First, a byte can be represented as a symbol in this field. Thereafter, this field is small enough to speed up some computing with discrete logarithm tables and at the same time sufficiently large to guarantee linear independence of the random coefficients with very high probability.

Encoded messages are linear combinations over $\mathbb{F}_{2^n}$ of multiple messages. This linear combination is not a concatenation: if the original messages are of size L, the encoded messages will be of size L too. An encoded message carries a part of the information of all the original messages, but not enough to recover any original message. After receiving enough encoded messages, the original messages will be decodable.

To perform the encoding, the sources must know n original messages defined as $M^1, ..., M^n$. Each time a source want to create an encoded message, it randomly chooses a sequence of coefficients $c_1, ..., c_n$, and computes the encoded message X as follows: $X = \sum_{i=1}^{n} c_i M^i$. An encoded message thus consists of a sequence of coefficients and the encoded information: (c, X).

Every participating node can recursively encode new messages from the one they received, including messages that have not been decoded. A node that received $(c^1, X^1), ..., (c^m, X^m)$ encoded messages, can encode a new message (c', X') encoded by choosing a random set of coefficients $d_1, ..., d_m$, computing the new encoded information $X' = \sum_{j=1}^{m} d_j X^j$ and computing the new sequence of coefficients $c'_i = \sum_{j=1}^{m} d_j c_i^j$.

An original message M^i can be considered as an encoded message by creating a coefficient vector $0, ..., 1, .., 0$ where 1 is at the ith position. The encoding of a message can therefore be considered as a subset of the recursive encoding technique.

Even if there is no theoretical limit on the number n of messages that can be encoded together, there are two reasons to limit it. First, Gauss-Jordan elimination has a $O(n^3)$ complexity, which becomes rapidly too expensive to compute. Then, the more the messages encoded together, the bigger the sequence of coefficients while the encoded information remains stable. In extreme cases this can result in sending mainly coefficients on the network instead of information. To encode more data, splitting messages in groups named generations solves the previous problems, as only messages in the same generation are encoded together.

However applying network coding to epidemic dissemination raises several challenges.

a) Assigning a message to a generation: Generations often consist of integers attached to a message. Messages with the same generation value are considered in the same generation and can be encoded between them. The value must be assigned in such a way than enough messages are in the same generation to benefit from RLNC properties but not too many to keep the decoding complexity sufficiently low and limit the size of the coefficients sent on the network. In a single source scenario, the size of the generation is a parameter of the protocol, and is determined by counting the number of messages sent in a given generation. However, with multiple sources, there is no way to know how many messages have been sent in a given generation.

b) Sending coefficients on the network: Coefficients are generally sent under the form of a dense vector over the network. Each value in the vector is linked to a message. On a single source scenario, that is not a problem, the source knows by advance how many messages it will send and can assign each message a position in the vector and start creating random linear combinations. In the case of multiple sources, the number of message is not available, and waiting to have enough messages to create a generation could delay message delivery and above all, the network traffic required to order the messages in the dense vector would ruin the benefits of network coding.

c) Pulling with RLNC: When doing pull-based rumor mongering, a node must have a way to ask what rumors it needs. Without network coding, it simply sends a message identifier to ask for a message. But sending a message identifier in the case of network coding raises several questions: does the node answer only if it has decoded the message? Or if it can generate a linear combination containing this message?

d) Estimating the number of missing packets: Some algorithms need to estimating the number of missing messages. Without network coding, the number of missing packets corresponds to the number of missing messages. But with RLNC, it is possible to have some linear combination for a given set of messages but without being able to decode them. All the messages are considered as missing but one packet could be enough to decode everything.

IV. Contribution

A. System model

Our model consists of a network of n nodes that all run the same program. These nodes communicate over a unicast unreliable and fully connected medium, such as UDP over Internet. Nodes can join and leave at any moment, as no graceful leave is needed, crashes are handled like departures. We consider that each nodes can obtain the address of some other nodes of the service via a Random Peer Sampling service [25]. There is no central authority to coordinates nodes, all operations are fully decentralized and all exchanges are asynchronous. We use the term message to denote the payload that must be disseminated to every node of the network, and the term packet to denote the exchanged content between two nodes. We consider the exchange of multiple messages and any node of the network can inject messages in the network, without any prior coordination between nodes (messages are not pre-labeled).

B. Solving RLNC challenges

Due to our multiple independent source model and our goal to cover push and pull protocols, we propose new solutions to the previously stated challenges:

a) Assigning generations with Lamport timestamps: With multiple senders, we need to find a rule that is applicable only with the knowledge of a single node when assigning a generation as relying on network would be costly, slow and unreliable. We chose Lamport timestamps [26] to delimit generations by grouping every messages with the same clock on the same generation. This method doesn't involve sending new packets as the clock is piggybacked on every network coded packet. When a node wants to disseminate a message,

it appends its local clock to the packet and update it, when it receives a message, it uses its local clock and the message clock to update its clock. This efficiently associated messages that are disseminated at the same time into the same generation.

b) Sending coefficients in sparse vectors: When multiple nodes can send independent messages, they have no clue on which identifiers are assigned by the other nodes. Consequently, they can only rely on their local knowledge to choose their identifiers. Choosing identifiers randomly on a namespace where all possible identifiers are used would lead to conflicts. That is why we decided to use a bigger namespace, where conflict probabilities are negligible when identifiers are chosen randomly. On a namespace of this size, it is impossible to send a dense vector over the network, however we can send a sparse vector: instead of sending a vector $c^1, ..., c^n$, we send m tuples, corresponding to the known messages, containing the message id and their coefficient: $(id(M^{i_1}), c^{i_1}), ..., (id(M^{i_m}), c^{i_m})$.

c) Pulling generations instead of messages: A node sends a list of generations that it has not fully decoded to its neighbors. The target node answers with one of the generation it knows. To determine if the information will be redundant, there is no other solution than asking the target node to generate a linear combination and try to add it to the node's local matrix. During our tests, it appears that blindly asking generations didn't increase the number of redundant packets compared to a traditional Push-Pull algorithm asking for a message identifier list while greatly decreasing message sizes.

d) Count needed independent linear combinations: To provide adaptiveness, estimating the number of useful packets needed to receive all the missing messages is needed by some protocols. Without network coding, the number of needed packets corresponds to the number of missing messages. With network coding, partially decoded packets are also considered as missing messages, but to decode them we need fewer packets than missing messages. In this case, the number of useful packets needed corresponds to the number of independent linear combinations needed.

C. CHEPIN

To ease the integration of RLNC in gossip-based dissemination algorithms, we encapsulated some common logic in algorithm 1. We represent a network-coded packet by a triplet: $\langle g, c, e \rangle$, where g is the generation number, c an ordered set containing the network coding coefficients and e the encoded payload.

We define 3 global sets: rcv, ids and dlv. rcv contains a list of network coded packets as described before that are modified each time a new one is received to stay linear independent until all messages are decoded. ids contains a list of known messages identifiers under the form $\langle g, gid \rangle$ where g is the generation and gid is the identifier of the message inside the generation. By using this tuple as unique identifier, we can reduce the number of bytes of gid as the probability of collision inside a generation is lower than the one in the whole system. Finally dlv contains a list of message identifiers similar to ids, but contains only identifiers of decoded messages.

Algorithm 1 Process RLNC Packets

```
1:  g ← 0                                          ▷ Encoding generation
2:  rcv ← SET()                                    ▷ Received packets
3:  ids ← ORDEREDSET()                             ▷ Known message identifiers
4:  dlv ← ORDEREDSET()                             ▷ Delivered message identifiers

5:  function PROCESSPACKET(p)
6:      if p = ∅ then
7:          return False

8:      ⟨g¹, c¹, _⟩ ← p
9:      oldRank ← RANK(g¹, rcv)
10:     rcv ← SOLVE(g¹, rcv ∪ {p})

11:     if oldRank = RANK(g¹, rcv) then
12:         return False                           ▷ Packet was useless

13:     for all ⟨id, _⟩ ∈ c¹ do
14:         ids ← ids ∪ {⟨g¹, id⟩}                 ▷ Register new identifiers

15:     for all ⟨g², c², e⟩ ∈ rcv do
16:         ⟨id, _⟩ ← c²[0]
17:         if g¹ = g² ∧ LEN(c²) = 1 ∧ ⟨g², id⟩ ∉ dlv then
18:             dlv ← dlv ∪ {⟨g², id⟩}
19:             DELIVER(e)                         ▷ New decoded message

20:     if g¹ > g ∨ RANK(g, rcv) ≥ 1 then
21:         g ← MAX(g¹, g) + 1                     ▷ Update Lamport Clock
22:     return True                                ▷ Packet was useful
```

The presented procedure relies on some primitives. RANK returns the rank of the generation, by counting the number of packets associated to the given generation. SOLVE returns a new list of packets after applying a Gaussian elimination on the given generation and removing redundant packets. DELIVER is called to notify a node of a message (if the same message is received multiple time, it is delivered only once).

This procedure updates the 3 global sets previously defined, delivers decoded messages and return the usefulness of the given packet. Internally, the node starts by adding the packet to the matrix and do a Gaussian elimination on the packet's generation (line 10), if decoding the packet didn't increase the matrix rank, the packet was useless and the processing stops here. Otherwise, the node must add unknown message identifiers from the packet coefficient list to the known identifiers set. After that, the node delivers all decoded messages thanks to the received packet and stores their identifiers in dlv. Finally, the node checks if the clock must be updated.

Algorithms 2 and 3 show how the above procedures can be used to implement push and pull gossip protocols. For push, we do not directly forward the received packet, but

instead forward a linear combination of the received packet's generation after adding it to our received packet list. For Pull, we request generations instead of messages. Like existing protocols, we keep a $rotation$ variable that rotates the set of missing identifiers, allowing missing generations to be generated in a different order on the next execution of the code block.

Algorithm 2 Push

```
1: k, ittl ← ...                          ▷ Push fanout and initial TTL
2: dottl, dodie ← ...                     ▷ Push strategy

3: function SendToNeighbourgs(h, headers)
4:     for k times do
5:         p ← Recode(h, rcv)
6:         Send(Push, p, headers)

7: function Broadcast(m, headers)
8:     id ← UniqueID()
9:     p ← ⟨g, {⟨id, 1⟩}, m⟩
10:    dlv ← dlv ∪ {⟨g, id⟩}
11:    ProcessPacket(p)
12:    headers.ttl ← ittl
13:    SendToNeighbourgs(g, headers)

14: function NCPush(p, headers)
15:    ⟨h, _, _⟩ ← p
16:    if ProcessPacket(p) ∨ ¬dodie then
17:        if dottl ∧ headers.ttl ≤ 0 then
18:            return
19:        if dottl then
20:            headers.ttl ← headers.ttl − 1
21:        SendToNeighbourgs(h, headers)
```

V. Application to Pulp

To apply our network-coding approach to a concrete use case, we design CHEPIN-Pulp, a protocol inspired by Pulp [14]. Pulp achieves cost-effective dissemination by optimizing the combination of push-based and pull-based gossip. In particular, nodes disseminate each message through a push-phase with little redundancy due to a fanout and a TTL configured to reach only a small portion of the network. As the push-phase doesn't provide a complete dissemination, the message will be retrieved by the rest of the network during a pull phase. To this end, each node periodically sends its list of missing messages to a random node. The target node answers with the first message it knows. To discover missing messages, nodes piggyback the list of recently received messages on every packet exchange. To improve reactivity and reduce delays, Pulp provides a pulling frequency-adaptation mechanism based on the node's estimation of missing messages and usefulness of its pull requests.

Algorithm 3 Pull

```
1: rotation ← 0                           ▷ Rotation position

2: function NCPullThread(headers)
3:     ask ← OrderedSet()
4:     rotation ← rotation + 1 mod |ids \ dlv|
5:     for all m ∈ Rotate(ids \ dlv, rotation) do
6:         ask ← ask ∪ Gen(m, rcv)
7:     Send(Pull, ask, headers)

8: function NCPull(asked, headers)
9:     p ← ∅
10:    if ∃ g ∈ asked, Rank(g, rcv) > 0 then
11:        p ← Recode(g, rcv)
12:    Send(PullReply, p, headers)

13: function NCPullReply(p)
14:    ProcessPacket(p)
```

On top of the two previously defined algorithms 2 and 3, we propose a push-pull algorithm inspired by Pulp (Algorithm 4. First, we must convert the Pulp message discovery mechanism which consists on exchanging recent message history via a trading window. The trading window is generated by the GetHeaders function, which will be added to every packets. On reception, the trading window will be retrieved and its new identifiers will be added to the ids set. The major difference with Pulp is that we don't trade identifiers of delivered messages but any identifiers we know, even if we compare both approaches in Section VI.

The adaptation mechanism is the second feature of Pulp, the pull frequency is adapted according to the number of missing packets and the usefulness of the pull requests. Our only modification is made on how to compute the number of missing packets, as we retained the number of needed independent linear combinations instead of the number of missing messages. To do so, we compute the difference between the number of messages identifiers and the number of independent linear combinations we have.

VI. Evaluation

We evaluated our solution in the Omnet++ simulator, using traces from PlanetLab and Overnet to simulate respectively the latency and the churn. To assess the effectiveness of CHEPIN, we implemented a modified version of Pulp as described in Section V, and compare it with the original Pulp protocol.

A. Experimental setup

We run our experiments with 1 000 nodes, sending 1 000 messages at a rate of 150 messages per second. Each message weighs 1KB and has a unique identifier. For Pulp, it is simply an integer encoded on 8 bytes. In the case of CHEPIN-Pulp, the unique identifier is composed of its generation encoded on

Algorithm 4 Pulp NC

1: $ts, sm \leftarrow ...$ ▷ Trading window size and margin
2: $\Delta_{adjust}, \Delta_{pull_{min}}, \Delta_{pull_{max}} \leftarrow ...$ ▷ Periods config.
3: $dottl, dodie \leftarrow True, True$ ▷ Set push strategyy
4: $\Delta_{pull} \leftarrow \Delta_{adjust}$

5: **function** GETHEADERS
6: $start \leftarrow \text{MAX}(0, |ids| - sm - tm)$
7: $end \leftarrow \text{MAX}(0, |ids| - sm)$
8: **return** $\{tw : ids[start : end]\}$

9: **upon receive** PUSH($p, headers$)
10: $ids \leftarrow ids \cup headers.tw$
11: NCPUSH(p, GETHEADERS())

12: **upon receive** PULL($asked, headers$)
13: $ids \leftarrow ids \cup headers.tw$
14: NCPULL($asked$, GETHEADERS())

15: **upon receive** PULLREPLY($p, headers$)
16: $ids \leftarrow ids \cup headers.tw$
17: NCPULLREPLY(p)

18: **thread** every Δ_{pull}
19: NCPULLTHREAD(GETHEADERS())

20: **thread** every Δ_{adjust}
21: $missingSize \leftarrow |ids| - |rcv|$
22: **if** $missingSize > prevMissingSize$ **then**
23: $\Delta_{pull} = \frac{\Delta_{adjust}}{missingSize - prevMissingSize + prev_{useful}}$
24: **else if** $missingSize > 0 \wedge prev_{useless} \leq prev_{useful}$ **then**
25: $\Delta_{pull} \leftarrow \Delta_{pull} \times 0.9$
26: **else**
27: $\Delta_{pull} \leftarrow \Delta_{pull} \times 1.1$
28: $\Delta_{pull} \leftarrow \text{MAX}(\Delta_{pull}, \Delta_{pull_{min}})$
29: $\Delta_{pull} \leftarrow \text{MIN}(\Delta_{pull}, \Delta_{pull_{max}})$
30: $prev_{useless} \leftarrow 0$
31: $prev_{useful} \leftarrow 0$
32: $prevMissingSize \leftarrow missingSize$

4 bytes and a unique identifier in this generation, also encoded on 4 bytes. Δ_{adapt} is set to 125 ms.

In order to accurately simulate latency, we use a PlanetLab trace [27]. Latency averages at 147 ms for a maximum of almost 3 seconds. Most of the values (5th percentile and 95th percentile) are between 20 ms and 325 ms. Finally we have a long tail of values between 325 ms and the maximum value.

B. Configuring the Protocols

To configure protocols, we chose an experimental approach. First, we selected a suitable value for the size of the trading window. As explained in Section V, too small values of this parameter result in wasted pull requests, and missing messages, while too large ones lead to wasted bandwidth. We therefore tested the ability of the original Pulp, and of our solution to achieve complete dissemination (i.e. all messages reach all nodes) with different trading window sizes, and a safety margin of 10. Results, not shown for space reasons, show that our solutions reaches complete dissemination with trading window sizes of at least 6, while Pulp requires trading-window sizes of at least 9. For the rest of our analysis, we therefore considered a trading-window size of 9, and a safety margin of 10. Nonetheless, this first experiment already hints at the better efficiency of our network-coding-based solution.

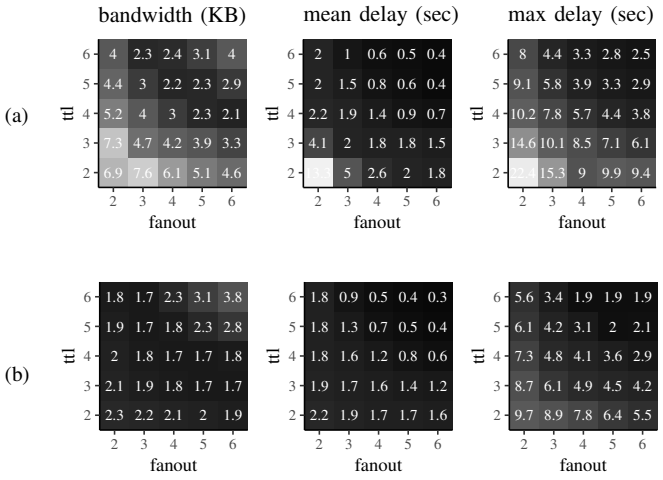


Fig. 2: Pulp (2a) and our algorithm (2b) behavior under various configuration of the protocol (fanout and time to live)

Next, we selected values for fanout and TTL. Figure 2 reports the delivery delays and bandwidth consumption of the two protocols with several values of these two parameters. To measure bandwidth consumption, we consider the ratio between the average amount of bandwidth consumed by the protocol, and the lower bound represented by the bandwidth required for the same task in a tree structure in a stable network. First, we observe that in terms of delays and bandwidth used, our network coding variant is more stable than the original Pulp. That is, with low values of fanout and ttl, the original algorithm deteriorates faster.

Next, we see that our network coding variant performs better or similarly for every combination of fanout and ttl both in terms of sent bandwidth and delay. The best configuration in term of sent data for Pulp corresponds to the configuration $k = 6, ttl = 4$ with 2.12 KB for 1KB of useful data and an average of 0.67 seconds to disseminate a message. Our network-coding solution reduces delay to 0.55, with a bandwidth consumption of 1.83KB/msg. With a fanout of 5 our solution further decreases consumed bandwidth to 1.66 KB/msg but with a slight increase in delay (0.83 s). Clearly, to achieve the minimum delays, the best strategy consists in boosting the push phase by increasing the TTL, but this defeats

the bandwidth-saving goal of Pulp and our approach. As a result, we use the configuration with $k = 6, ttl = 4$ for both protocols in the rest of our comparison.

C. Bandwidth and delay comparison

We evaluate how our algorithm performs over time in Figure 3. First, we logged the number of packets sent per second for the three types of packets: push, pull and pull reply. As we configured the two protocols with the same fanout and TTL, we would expect seeing almost the same number of push packets. But our network-coded variant sends 12% more push packets. Pulp stops forwarding a push packet if the corresponding message is already known. But since our variant can use a large number of linear combinations, our algorithm manages to exploit the push-phase better, thereby reducing the number of packets sent in the pull phase: 33% fewer pull and pull reply packets. This strategy enables us to have a packet ratio of only 2.27 instead of 2.70.

As network coded packets include a sparse vector containing messages identifiers and values, our the algorithm has larger pull and pull reply packets than Pulp. Considering push packets, we also send more of them, which explains why we send 17% more data for these packets. However, as our pull phase is way smaller than the one from Pulp, the pull reply packets of our algorithm consume 28% less data than the ones from Pulp. We can also notice that the pull packets consume less data than the two others. Indeed, these packets never contain the 1 KB payload. However, we can notice that our algorithm still consumes less data than Pulp as we transmit generation id instead of each message id. With 150 messages/second, at a given time time, every nodes will be aware of a huge list of missing messages and ask it to their peers. Generally, this list will contain messages sent approximately at the same time, so probably in the same generation, which explain why it is way more efficient to ask for generation identifiers, and why pull packets will be smaller for our algorithm. These two facts enable us to have a data ratio 1.84 instead 2.12.

Finally, we study the distribution delay of each message. As our algorithm has a longer push phase, delays are smaller on average. We see a downward slope pattern on our algorithm's delays, especially on the maximum delays part. This pattern can be explained by the fact that decoding occurs at the end of each generation, so messages that are sent earlier wait for longer than the most recent ones.

D. Adaptiveness optimization

We now carry out a sensitivity analysis to understand the reasons for our improved performance. To this end, Figure 4 compares our algorithm with to Pulp and with two intermediate variants.

The first variant corresponds to a modification in the GET-TRADINGWINDOW function of algorithm 1. Instead of using the message identifiers contained in the ids variable, we use the message identifiers contained in the dlv variable like in the case of the standard Pulp protocol. In other words, we disseminate the identifiers of messages we have decoded and not those we are aware of.

The second variant is a modification on how we count the number of missing messages at line 21 in algorithm 4. For this variant, we do $missingSize \leftarrow |missing|$ like in the original pulp. We thus evaluate the number of missing messages by counting all the message identifiers we have not yet decoded, without taking into account the progress of the decoding in our generations.

The two variants perform worse than our solution both in terms of delay and bandwidth. Variant 1 does not manage to achieve complete dissemination with a fanout of 6 and a TTL of 4, while Variant 2 achieves complete dissemination but at a higher cost ratio: 2.4 instead of 1.83 for our solution. This shows the importance of the modifications we made to the Pulp protocol.

To understand how these modifications behave, Figure 4 shows the different between the number of useful and useless packets. We see that our algorithm and Variant 1 perform similarly. The Pulp algorithm has a more efficient pull strategy than ours, possibly because there are more messages to pull, we receive redundant linear combination for the requested generation or the frequency adaptation is not optimal in our case. However, we see that we obtain way better results than the second variant, which mainly pull useless messages. At the end, we see lot of useless messages, as we don't inject useful messages anymore. The adaptive part of the algorithm therefore decreases the number of pull per second, to reduce useless messages.

When we look at the pull period, we see that Variant 1 and our algorithm are quite similar, with the biggest pull period as they have less messages to pull. Pulp has a smaller pull period, but has it has more messages to pull, it is not surprising. Our second variant pull period is the smallest even though this variant hasn't more messages to pull than our algorithm or our first variant, and explain why we have many useless messages: we pull to frequently.

Finally, when we study the evolution of our missing message estimation, it appears that the first variant has the lowest estimation of missing messages. It is due to the fact that we relay only decoded messages via the trading window, which adds a delay and improve the chances to receive the information via network coding coefficients.

However, it appears that this method is not efficient to disseminate message identifiers as it fails to achieve a complete dissemination with high probability. Indeed, if a message is sent at the end of a generation, its content and existence will only be forwarded on its push phase and not by other messages of the generation.

E. Behaviour under network variation

Figure 5 shows how our algorithm performs under different network configurations. We see that when the number of messages per second decreases, the difference in term of data sent between the algorithms decreases too. This can be explained by the fact that our generations become smaller

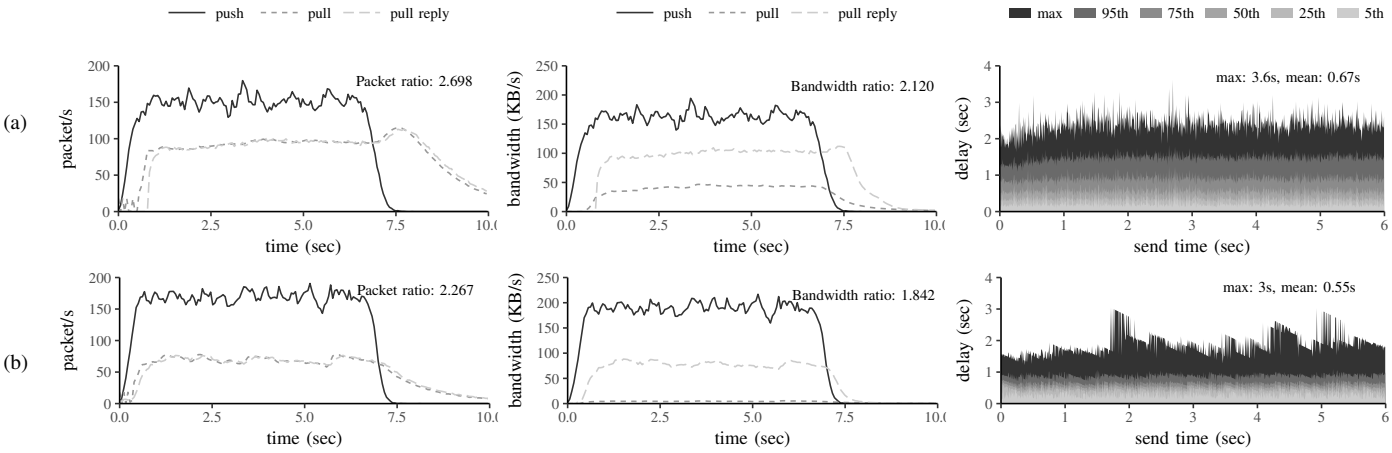
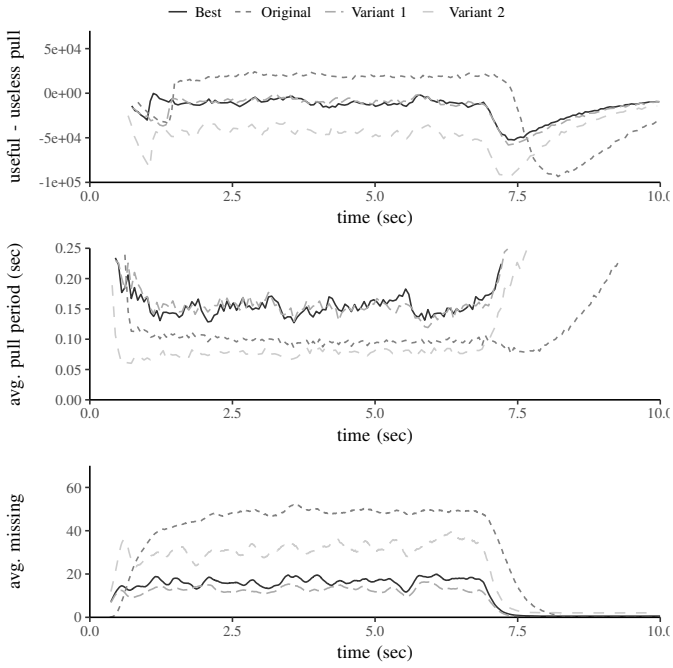


Fig. 3: Comparison of exchanged packet rate, used bandwidth and message delay for 3a pulp original ($k = 6, ttl = 4$) and 3b our algorithm ($k = 5, ttl = 4$)



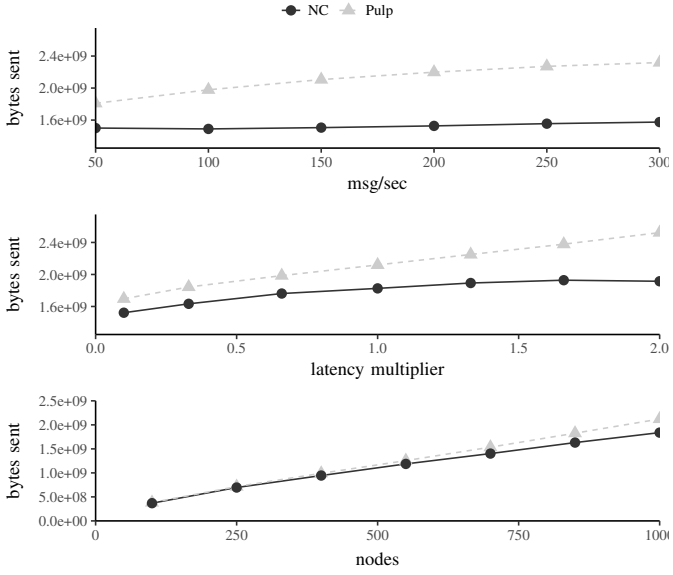


Fig. 5: Variation of network configuration

Fig. 4: How adaptiveness algorithms impact the protocol efficiency

when the number of messages per second decreases, and so less useful.

At an extreme, when we have only one message per generation, we have the same model as Pulp: asking for a list of generation identifiers is similar to asking a list of message identifiers in Pulp. The network coded packets will contain a generation identifier, a sparse vector containing only one message identifier and one coefficient and the message. As a generation identifier plus a message identifier of our algorithm has the same size of a message identifier in Pulp, the only overhead is the one byte coefficient value.

We use an Overnet trace to simulate churn [28]. The trace contains more than 600 active nodes over 900 with continuous churn—around 0.14143 events per second.

We use this trace replayed at different speed to evaluate the impact of churn on the delivery delays of our messages, as plotted on Figure 6. We chose to re-execute the trace at different speed: 500, 1000 and 2000 times faster for respectively 71, 141 and 283 churn events per second. We see that the original Pulp algorithm is not affected by churn, as the average and maximum delivery delays stay stable and similar to those without churn. Considering the average delay, it's also the case for our algorithm, where the average delay didn't evolve. The maximum delay doesn't evolve significantly either. However

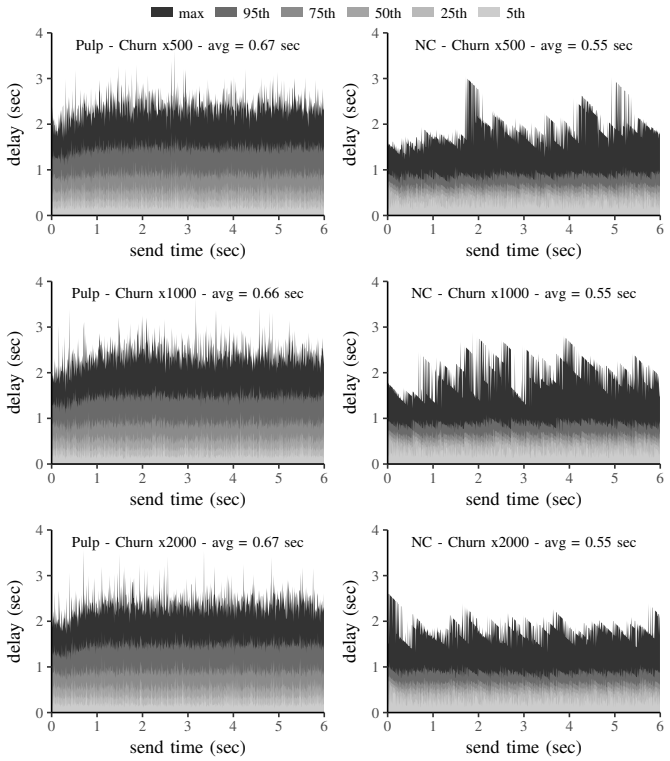


Fig. 6: Delay observation with churn on the system

we can see huge differences in the shape of the maximum delay for each individual messages. Indeed, the decoding order and the generation delimitation are affected by the churn, but limited impact on message dissemination.

VII. Conclusion

Distributed systems become more and more complex and dynamic with time. Due to their properties and resistance to failure, data dissemination protocols are the cornerstone of any distributed system. However, these advantages come at the cost of bandwidth spent by the embedded redundancy in the protocol. We introduce CHEPIN, a multi source gossip protocol that uses Lamport clocks to create generations, and sparse vectors to exchange coefficients. We have demonstrated that is possible to apply RLNC for push and pull algorithm by thoroughly evaluating our approach. At worst, CHEPIN performs like the state of the art. At best, CHEPIN significantly improves both delay and bandwidth consumption. As future work, we would like to investigate benefits of overlapping generations on message discovery and efficiency. We are also interested by improving CHEPIN's adaptiveness and extend its generation management.

Acknowledgment

We wish to thank the authors of [14] for providing useful information about their work. This work was partially funded by the O'Browser ANR grant (ANR-16-CE25-0005-03).

References

[1] M. Jelasity and Ö. Babaoglu, "T-man: Gossip-based overlay topology management," in *Engineering Self-Organising Systems, Third International Workshop, ESOA 2005, Utrecht, The Netherlands, July 25, 2005, Revised Selected Papers*, 2005, pp. 1–15.

[2] S. Bouget, Y.-D. Bromberg, A. Luxey, and F. Taïani, "Pleiades: Distributed Structural Invariants at Scale," in *DSN 2018*. Luxembourg, Luxembourg: IEEE, Jun. 2018, pp. 1–12.

[3] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[4] G. DeCandia, D. Hastorun, M. Jampani *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[5] A. Dadgar, J. Phillips, and J. Currey, "Lifeguard : Swim-ing with situational awareness," *CoRR*, vol. abs/1707.00788, 2017.

[6] A. Das, A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *In Proc. 2002 Intnl. Conf. Dependable Systems and Networks (DSN 02*, 2002, pp. 303–312.

[7] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, pp. 219–252, Aug. 2005.

[8] D. Frey, R. Guerraoui, A.-M. Kermarrec, and M. Monod, "Live Streaming with Gossip," Inria Rennes Bretagne Atlantique ; RR-9039, Research Report RR-9039, Mar. 2017.

[9] D. Frey, R. Guerraoui, A. Kermarrec, M. Monod, and V. Quéma, "Stretching gossip with live streaming," in *DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*, 2009, pp. 259–264.

[10] E. Androulaki, A. Barger, V. Bortnikov *et al.*, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," *CoRR*, vol. abs/1801.10228, 2018.

[11] B. Ndelec, P. Molli, and A. Mostefaoui, "CRATE: Writing Stories Together with Our Browsers," in *Proceedings of the 25th International Conference Companion on World Wide Web*. International World Wide Web Conferences Steering Committee, 2016, pp. 231–234.

[12] B. Koldehofe, "Simple gossiping with balls and bins." *Stud. Inform. Univ.*, vol. 3, no. 1, pp. 43–60, 2004.

[13] P. Euster, R. Guerraoui, A.-M. Kermarrec, and L. Maussoulie, "From epidemics to distributed computing," *IEEE Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[14] P. Felber, A.-M. Kermarrec, L. Leonini *et al.*, "Pulp: An adaptive gossip-based dissemination protocol for multi-source message streams," *Peer-to-Peer Networking and Applications*, vol. 5, no. 1, pp. 74–91, 2012.

[15] S. Deb, M. Mdard, and C. Choute, "Algebraic gossip: A network coding approach to optimal multiple rumor mongering," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. SI, pp. 2486–2507, 2006.

[16] B. Haeupler, "Analyzing network coding gossip made easy," in *Proceedings of the forty-third annual ACM symposium on Theory of computing*. ACM, 2011, pp. 293–302.

[17] A. Castaeda, S. Rajsbaum, and M. Raynal, "The renaming problem in shared memory systems: An introduction," *Computer Science Review*, vol. 5, no. 3, pp. 229–251, 2011.

[18] A. J. Demers, D. H. Greene, C. Hauser *et al.*, "Epidemic Algorithms for Replicated Database Maintenance," *Operating Systems Review*, vol. 22, no. 1, pp. 8–32, 1988.

[19] S. Sanghavi, B. E. Hajek, and L. Massouli, "Gossiping With Multiple Messages," *IEEE Trans. Information Theory*, vol. 53, no. 12, pp. 4640–4654, 2007.

[20] M.-L. Champel, A.-M. Kermarrec, and N. L. Scouarnec, "FoG: Fighting the Achilles' Heel of Gossip Protocols with Fountain Codes," in *SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, 2009, pp. 180–194.

[21] P. A. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Allerton Conference on Communication, Control, and Computing*, Oct. 2003.

[22] C. Fragouli, J. Widmer, and J. Y. L. Boudec, "Efficient Broadcasting Using Network Coding," *IEEE/ACM Transactions on Networking*, vol. 16, no. 2, pp. 450–463, Apr. 2008.

[23] S. Katti, H. Rahul, W. Hu *et al.*, "XORs in the air: Practical wireless network coding," in *ACM SIGCOMM computer communication review*, vol. 36. ACM, 2006, pp. 243–254.

[24] C. Fragouli, J.-Y. Le Boudec, and J. Widmer, "Network Coding: An Instant Primer," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 63–68, Jan. 2006.
[25] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based Peer Sampling," *TOCS*, vol. 25, no. 3, 2007.
[26] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[27] R. Zhu, B. Liu, D. Niu *et al.*, "Network Latency Estimation for Personal Devices: A Matrix Completion Approach," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 724–737, 2017.
[28] R. Bhagwan, S. Savage, and G. M. Voelker, "Understanding Availability," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.